Migrate Win32 C/C++ application to Linux on POWER, Part 2: Mutexes

Nam Keung
Chakarat Skawratananond

February 10, 2005

This series of articles helps you migrate your Win32 C/C++ applications to Linux on POWER. Senior programmer Nam Keung and pSeries® Linux technical consultant Chakarat Skawratananond illustrate how to map Win32 to Linux with respect to mutex application program interfaces (APIs). Part 1 of this series focused on Win32 API mapping.

Introduction

This article focuses on mutex primitives. You are encouraged to review the following sections in Part 1 of this series before continuing:

- Initialization
- Process
- Threads
- Shared memory

Mutexes

A mutex provides exclusive access control for a resource between threads, as shown in Table 1 below. It is a simple lock with only the thread that owns the lock being able to release the mutex. It ensures the integrity of a shared resource that they access (most commonly shared data), by allowing only one thread to access it at a time.

Table 1. Mutexes

Win32	Linux
`CreateMutex(0, FALSE, 0);`	`pthread_mutex_init(&mutex, NULL))`
`CloseHandle(mutex);`	`pthread_mutex_destroy(&mutex)`
`WaitForSingleObject(mutex, INFINITE))`	`pthread_mutex_lock(&mutex)`
`ReleaseMutex(mutex);`	`pthread_mutex_unlock(&mutex)`

Create a mutex

In Win NT/Win2K, all mutexes are recursive.

In Win32, `CreateMutex()` provides exclusive access control for a resource between threads within the current process. This method enables threads to serialize their access to the resources within a process. Once the mutual exclusion handle is created, itÂ#s available to all threads in the current process (see Listing 1 below).

Listing 1. Create a mutex

```
HANDLE CreateMutex(
 LPSECURITY_ATTRIBUTES lMutexAttributes,
 BOOL   lInitialOwner,
 LPCTSTR   lName
)
```

Linux uses the pthread library call, `pthread_mutex_init()`, to create the mutex, as shown in Listing 2 below.

Listing 2. pthread

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

Linux has three kinds of mutex. The mutex kind determines what happens if a thread attempts to lock a mutex it already owns in a pthread_mutex_lock:

Fast mutex:
: While trying to lock the mutex using the `pthread_mutex_lock()`, the calling thread is suspended forever.

Recursive mutex:
: `pthread_mutex_lock()` immediately returns with a success return code.

Error check mutex:
: `pthread_mutex_lock()` immediately returns with the error code EDEADLK.

The mutex kind can be set in two ways. Listing 3 illustrates a static way of setting a mutex.

Listing 3. Static way for setting a mutex

```
/* For Fast mutexes */
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

/* For recursive mutexes */
```

You can lock a mutex with the function: `pthread_mutex_lock(pthread_mutex_t *mutex)`. This function gets a pointer to the mutex it is trying to lock. The function returns when the mutex is locked, or if an error occurred. The error is not due to the locked mutex. The function waits until the mutex becomes unlocked.

Another way of setting the mutex kind is by using a mutex attribute object. To do this, `pthread_mutexattr_init()` is called to initialize the object followed by a `pthread_mutexattr_settype()`, which sets the kind of mutex, as shown in Listing 4 below.

Listing 4. Setting a mutex by attribute

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int kind);
```

A mutex is unlocked with the function (see Listing 5):

Here's the sample code for creating a mutex (see Listings 6 and 7 below).

Listing 5. The unlock fuction

```
pthread_mutex_unlock(pthread_mutex_t *mutex))
```

Listing 6. Win32 sample code

```
HANDLE mutex;

mutex = CreateMutex(0, FALSE, 0);
if (!(mutex))
{
    return RC_OBJECT_NOT_CREATED;
}
```

Listing 7. Equivalent Linux code

```
pthread_mutexattr_t  attr;
pthread_mutex_t      mutex;

pthread_mutexattr_init (&attr);
if (rc = pthread_mutex_init(&mutex, &attr))
{
    return RC_OBJECT_NOT_CREATED;
}
```

Destroying a mutex

In Win32, the `CloseHandle()` method (see Listing 8) deletes an object to provide an exclusive access control for a resource within a current process. After the deletion of the object, the mutex object is invalid until the `CloseHandle()` method initializes it again by calling `CreateMutex`.

Once there is no longer an exclusive access for a resource, you should destroy it by calling this method. If you need to relinquish the ownership of the object, the `ReleaseMutex()` method should be called.

The `pthread_mutex_destroy()` in Linux destroys a mutex object, which frees the resources it might hold. It also checks whether the mutex is unlocked at that time (see Listing 9).

Listing 8. Win32 sample code

```
if(WaitForSingleObject(mutex, (DWORD)0) == WAIT_TIMEOUT )
return RC_NOT_OWNER;

CloseHandle(mutex);
```

Listing 9. Linux code

```
if (pthread_mutex_destroy(&mutex) == EBUSY)
 return RC_NOT_OWNER;
```

Locking a mutex

In Win32, the `WaitForSingleObject()` (see Listing 10) blocks a request for exclusive access to a resource within the current process. A process can block a request in the following ways:

1. If a request for exclusive access to the resource is unlocked, this method locks it.
2. If the exclusive access to the resource is already locked, this method blocks the calling thread until it is unlocked.

Linux uses a `pthread_mutex_lock()` (see Listing 11).

You can also use the `pthread_mutex_trylock()` to test whether a mutex is locked without actually blocking it. If another thread locks the mutex, the `pthread_mutex_trylock` will not block. It immediately returns with the error code EBUSY.

Listing 10. Win32 sample code

```
if ((rc = WaitForSingleObject(mutex, INFINITE)) == WAIT_FAILED)
     return RC_LOCK_ERROR;
```

Listing 11. Linux code

```
if (rc = pthread_mutex_lock(&mutex))
return RC_LOCK_ERROR;
```

Releasing or unlocking a mutex

Win32 uses `ReleaseMutex()` (see Listing 12) to release exclusive access to a resource. This call might fail if the calling thread does not own the mutex object.

Linux uses `pthread_mutex_unlock()` to release or unlock the mutex (see Listing 13).

Listing 12. Win32 sample code

```
If (! ReleaseMutex(mutex))
{
 rc = GetLastError();
 return RC_UNLOCK_ERROR;
}
```

Listing 13. Linux code

```
if (rc = pthread_mutex_unlock(&mutex))
return RC_UNLOCK_ERROR;
```

Mutex sample codes

Here is the Win32 sample code to acquire a mutex within a process (see Listing 14):

Listing 14. Win32 sample code

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

void  thrdproc  (void *data); //the thread procedure (function) to be executed

HANDLE    mutex;

int main( int argc, char **argv )
{
      int        hThrd;
      unsigned   stacksize;
      HANDLE     *threadId1;
      HANDLE     *threadId2;
      int        arg1;
      DWORD   rc;

       if( argc < 2 )
  arg1 = 7;
 else
  arg1 = atoi( argv[1] );

 printf( "Intra Process Mutex test.\n" );
 printf( "Start.\n" );
       mutex = CreateMutex(0, FALSE, 0);
      if (mutex==NULL)
            return RC_OBJECT_NOT_CREATED;

 printf( "Mutex created.\n" );

      if ((rc = WaitForSingleObject(mutex, INFINITE)) == WAIT_FAILED)
             return RC_LOCK_ERROR ;

 printf( "Mutex blocked.\n" );


      if( stacksize < 8192 )
            stacksize = 8192;
        else
            stacksize = (stacksize/4096+1)*4096;

      hThrd = _beginthread( thrdproc, // Definition of a thread entry
                                 NULL,
                           stacksize,
                          "Thread 1");

      if (hThrd == -1)
            return RC_THREAD_NOT_CREATED);

      *threadId1 = (HANDLE) hThrd;

      hThrd = _beginthread( thrdproc, // Definition of a thread entry
                                 NULL,
                           stacksize,
                           Thread 2");

      if (hThrd == -1)
            return RC_THREAD_NOT_CREATED);

      *threadId2 = (HANDLE) hThrd;

 printf( "Main thread sleeps 5 sec.\n" );

 Sleep( 5*1000 );
```

```
      if (! ReleaseMutex(mutex))
      {
          rc = GetLastError();
          return RC_UNLOCK_ERROR;
      }

 printf( "Mutex released.\n" );
 printf( "Main thread sleeps %d sec.\n", arg1 );

 Sleep( arg1 * 1000 );

       if( WaitForSingleObject(mutex, (DWORD)0) == WAIT_TIMEOUT )
             return RC_NOT_OWNER;

       CloseHandle(mutex);

 printf( "Mutex deleted. (%lx)\n", rc );
 printf( "Main thread sleeps 5 sec.\n" );

 Sleep( 5*1000 );
 printf( "Stop.\n" );
 return 0;
}

void thread_proc( void *pParam )
{
       DWORD rc;

 printf( "\t%s created.\n", pParam );
       if ((rc = WaitForSingleObject(mutex, INFINITE)) == WAIT_FAILED)
              return RC_LOCK_ERROR;

 printf( "\tMutex blocked by %s. (%lx)\n", pParam, rc );
 printf( "\t%s sleeps for 5 sec.\n", pParam );

 Sleep( 5* 1000 );

       if (! ReleaseMutex(mutex))
       {
          rc = GetLastError();
          return RC_UNLOCK_ERROR;
       }
 printf( "\tMutex released by %s. (%lx)\n", pParam, rc );
}
```

An equivalent Linux sample code to acquire mutex within a process (see Listing 15):

Listing 15. Equivalent Linux sample code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>

void  thread_proc (void * data);

pthread_mutexattr_t     attr;
pthread_mutex_t    mutex;

int main( int argc, char **argv )
{
      pthread_attr_t              pthread_attr;
      pthread_attr_t              pthread_attr2;
      pthread_t             threadId1;
      pthread_t                   threadId2;
```

```
     int                     arg1;
     int              rc = 0;

if( argc < 2 )
 arg1 = 7;
else
 arg1 = atoi( argv[1] );

printf( "Intra Process Mutex test.\n" );
printf( "Start.\n" );
pthread_mutexattr_init( &attr );
if ( rc = pthread_mutex_init( &mutex, NULL))
      {
   printf( "Mutex NOT created.\n" );
         return RC_OBJECT_NOT_CREATED;
      }
printf( "Mutex created.\n" );
      if (rc = pthread_mutex_lock (&mutex))
      {
   printf( "Mutex LOCK ERROR.\n" );
         return RC_LOCK_ERROR;
      }
printf( "Mutex blocked.\n" );

      if (rc = pthread_attr_init(&pthread_attr))
      {
   printf( "pthread_attr_init ERROR.\n" );
         return RC_THREAD_ATTR_ERROR;
      }

if (rc = pthread_attr_setstacksize(&pthread_attr, 120*1024))
      {
   printf( "pthread_attr_setstacksize ERROR.\n" );
         return RC_STACKSIZE_ERROR;
      }

if (rc = pthread_create(&threadId1,
                        &pthread_attr,
           (void*(*)(void*))thread_proc,
                           "Thread 1" ))
      {
   printf( "pthread_create ERROR.\n" );
         return RC_THREAD_NOT_CREATED;
      }

      if (rc = pthread_attr_init(&pthread_attr2))
      {
   printf( "pthread_attr_init2 ERROR.\n" );
         return RC_THREAD_ATTR_ERROR;
      }

if (rc = pthread_attr_setstacksize(&pthread_attr2, 120*1024))
      {
   printf( "pthread_attr_setstacksize2 ERROR.\n" );
         return RC_STACKSIZE_ERROR;
      }

if (rc = pthread_create(&threadId2,
                         &pthread_attr2,
           (void*(*)(void*))thread_proc,
                           "Thread 2" ))
      {
   printf( "pthread_CREATE ERROR2.\n" );
   return RC_THREAD_NOT_CREATED;
      }

printf( "Main thread sleeps 5 sec.\n" );
```

```
 sleep (5);

       if (rc = pthread_mutex_unlock(&mutex))
       {
    printf( "pthread_mutex_unlock ERROR.\n" );
    return RC_UNLOCK_ERROR;
       }

 printf( "Mutex released.\n" );
 printf( "Main thread sleeps %d sec.\n", arg1 );
 sleep(arg1);

 pthread_mutex_destroy(&mutex);

       printf( "Main thread sleeps 5 sec.\n" );
 sleep( 5 );
 printf( "Stop.\n" );
 return 0;
}

void thread_proc( void *pParam )
{
       int nRet;

 printf( "\t%s created.\n", pParam );
 if (nRet = pthread_mutex_lock(&mutex))
       {
  printf( "thread_proc Mutex LOCK ERROR.\n" );
  return RC_LOCK_ERROR;
       }
 printf( "\tMutex blocked by %s. (%lx)\n", pParam, nRet );
 printf( "\t%s sleeps for 5 sec.\n", pParam );
 sleep(5);
       if (nRet = pthread_mutex_unlock(&mutex))
       {
    printf( " thread_proc :pthread_mutex_unlock ERROR.\n" );
    return RC_UNLOCK_ERROR;
       }

 printf( "\tMutex released by %s. (%lx)\n", pParam, nRet );
}
```

Here is another Win32 sample code to acquire mutex between processes.

Mutexes are system-wide objects which multiple processes can see. If Program A creates a mutex, Program B can see that same mutex. Mutexes have names, and only one mutex of a given name can exist on a machine at a time. If you create a mutex called "My Mutex", no other program can create a mutex with that name, as shown in Listings 16 and 18 below.

Listing 16. Win32 inter process mutex sample code Process 1

```
#include <stdio.h>
#include <windows.h>

#define WAIT_FOR_ENTER  printf( "Press ENTER\n" );getchar()

int main()
{
      HANDLE mutex;
      DWORD   rc;

      printf( "Inter Process Mutex test - Process 1.\n" );
      printf( "Start.\n" );
```

```
      SECURITY_ATTRIBUTES    sec_attr;

      sec_attr.nLength              = sizeof( SECURITY_ATTRIBUTES );
      sec_attr.lpSecurityDescriptor = NULL;
      sec_attr.bInheritHandle       = TRUE;

      mutex = CreateMutex(&sec_attr, FALSE, "My Mutex");
      if( mutex == (HANDLE) NULL )
          return RC_OBJECT_NOT_CREATED;

      printf( "Mutex created.\n" );

      WAIT_FOR_ENTER;

      if ( WaitForSingleObject(mutex, INFINITE) == WAIT_FAILED)
           return RC_LOCK_ERROR;

 printf( "Mutex blocked.\n" );
 WAIT_FOR_ENTER;

      if( ! ReleaseMutex(mutex) )
      {
            rc = GetLastError();
            return RC_UNLOCK_ERROR;
      }

       printf( "Mutex released.\n" );

       WAIT_FOR_ENTER;

 CloseHandle (mutex);

 printf( "Mutex deleted.\n" );
 printf( "Stop.\n" );

 return OK;
}
```

In here, the System V Interprocess Communications (IPC) functions are used for Linux implementation, as shown in Listings 17 and 19.

Listing 17. Equivalent Linux sample code Process 1

```
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define WAIT_FOR_ENTER    printf( "Press ENTER\n" );getchar()

union semun {
    int                 val;   /* value for SETVAL             */
    struct semid_ds    *buf;   /* buffer for IPC_STAT, IPC_SET */
    unsigned short     *array; /* array for GETALL, SETALL     */
    struct seminfo     __buf;  /* buffer for IPC info          */
};

main()
{
      int        shr_sem;
      key_t         semKey;
      struct sembuf   semBuf;
      int  flag;
      union semun      arg;
```

```
printf( "Inter Process Mutex test - Process 1.\n" );
printf( "Start.\n" );

flag = IPC_CREAT;

if( ( semKey = (key_t) atol( "My Mutex" ) ) == 0 )
      return RC_INVALID_PARAM;

flag |= S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP;

shr_sem  = (int) semget( semKey, 1, flag );

if (shr_sem < 0)
 return RC_OBJECT_NOT_CREATED;

      arg.val = 1;
if (semctl(shr_sem, 0, SETVAL, arg) == -1)
 return RC_OBJECT_NOT_CREATED;

printf( "Mutex created.\n" );

WAIT_FOR_ENTER;

      semBuf.sem_num = 0;
      semBuf.sem_op = -1;
      semBuf.sem_flg = SEM_UNDO;
      if (semop(shr_sem, &semBuf, 1) != 0)
               return RC_LOCK_ERROR;

printf( "Mutex blocked.\n" );

      WAIT_FOR_ENTER;

      semBuf.sem_num = 0;
      semBuf.sem_op  = 1;
      semBuf.sem_flg = SEM_UNDO;

      if (semop(shr_sem, &semBuf, 1) != 0)
          return RC_UNLOCK_ERROR;

printf( "Mutex released.\n" );

WAIT_FOR_ENTER;

      semctl( shr_sem, 0, IPC_RMID );

printf( "Mutex deleted.\n" );
printf( "Stop.\n" );

return 0;
```

Listing 18. Win32 inter process mutex sample code Process 2

```
#include <stdio.h>
#include <windows.h>


int main()
{
      HANDLE      mutex;

      printf( "Inter Process Mutex test - Process 2.\n" );
      printf( "Start.\n" );

      SECURITY_ATTRIBUTES           sec_attr;
```

```
      sec_attr.nLength              = sizeof( SECURITY_ATTRIBUTES );
      sec_attr.lpSecurityDescriptor = NULL;
      sec_attr.bInheritHandle       = TRUE;

      mutex = OpenMutex(MUTEX_ALL_ACCESS, TRUE, Â#My Mutex");
      if( mutex == (HANDLE) NULL )
          return RC_OBJECT_NOT_CREATED;

 printf( "Mutex opened. \n");
 printf( "Try to block mutex.\n" );

      if ( WaitForSingleObject(mutex, INFINITE) == WAIT_FAILED)
         return RC_LOCK_ERROR;

 printf( "Mutex blocked. \n" );
 printf( "Try to release mutex.\n" );

      if( ! ReleaseMutex(mutex) )
            return RC_UNLOCK_ERROR;

      printf( "Mutex released.\n" );

      CloseHandle (mutex);

      printf( "Mutex closed. \n");
      printf( "Stop.\n" );

      return OK;
}
```

Listing 19. Equivalent Linux sample code Process 2

```
#include <stdio.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <unistd.h>

int main()
{
      int             mutex;
      key_t           semKey;
      struct sembuf   semBuf;
      int             flag;
      int        nRet=0;

      printf( "Inter Process Mutex test - Process 2.\n" );
      printf( "Start.\n" );

      flag = 0;

      if( ( semKey = (key_t) atol( "My Mutex" ) ) == 0 )
            return RC_INVALID_PARAM;

      flag |= S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP;

      mutex = (int) semget( semKey, 1, flag );

      if (mutex == -1)
          return RC_OBJECT_NOT_CREATED;

      printf( "Mutex opened \n");
      printf( "Try to block mutex.\n" );

      semBuf.sem_num = 0;
      semBuf.sem_op = -1;
```

```
      semBuf.sem_flg = SEM_UNDO;
      if (semop(mutex, &semBuf, 1) != 0)
          return RC_LOCK_ERROR;

      printf( "Mutex blocked. \n");
      printf( "Try to release mutex.\n" );

      semBuf.sem_num = 0;
      semBuf.sem_op  = 1;
      semBuf.sem_flg = SEM_UNDO;
      if (semop(mutex, &semBuf, 1) != 0)
          return RC_UNLOCK_ERROR;

      printf( "Mutex released. \n");

      printf( "Mutex closed. \n");
      printf( "Stop.\n" );

      return 0;
}
```

Conclusion

In this article, we covered the mapping of Win32 to Linux with respect to mutex APIs. We also referenced lists of mutex sample codes to help you when you undertake the migration activity involving Win32 to Linux. The next article in this series will cover semaphores.

Notice updates

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Related topics

- Read other installments in the *Migrate Win32 C/C++ applications to Linux on POWER* series:
  - Process, thread, and shared memory services
- Want more? The developerWorks eServer zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on the eServer brand.